Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Artificial Intelligence 153 (2004) 307–337

Artificial Intelligence

www.elsevier.com/locate/artint

Heuristic planning: A declarative approach based on strategies for action selection

Josefina Sierra-Santibáñez

Dept. Lenguajes y Sistemas Informáticos, Universidad Politécnica de Cataluña, E-08034 Barcelona, Spain

Received 14 November 2001

Abstract

This paper introduces the notion of *heuristic planning*, and describes a particular approach to heuristic planning based on a *declarative formalization of strategies for action selection*. This approach is compared with some heuristic planning systems proposed in the literature. The heuristic information and declarative formalisms for the representation of heuristic knowledge used by these systems are compared in terms of their capacity of controlling the search process and their effectiveness for solving some planning problems. Finally, the results of some experiments on *heuristic answer set planning* are described in order to show how heuristics can be used to improve the scalability of answer set planning.

Keywords: Heuristic planning; Declarative formalizations; Action selection strategies; Answer set planning

1. Introduction

In the paper *Programs with Common Sense* [27], John McCarthy described a program called *the advice taker* which would represent its knowledge declaratively and would accept advice from its users in the form of declarative sentences. This paper gave raise to the *logical approach to Artificial Intelligence*. A major drawback of current logical reasoning systems is their inability to use domain and problem dependent heuristic advice to guide their search for solutions. In this paper, we address this problem by presenting a scheme for the declarative formalization of heuristics and showing how a general purpose forward chaining planner can use declarative formalizations of heuristics to improve its performance in several domains.

In particular, we introduce the notion of heuristic planning and present a particular approach to heuristic planning based on a declarative formalization of strategies for action

E-mail address: jsierra@lsi.upc.es (J. Sierra-Santibáñez).

0004-3702/$ – see front matter
doi:10.1016/j.artint.2002.11.001

selection described in [38], which illustrates the idea of the declarative formalization of heuristics proposed by John McCarthy in [27]. This approach is compared with the proposals of [4,37] and [9]. The heuristic information and declarative formalisms for the representation of heuristic knowledge used by these systems are compared in terms of their capacity of controlling the search process and their effectiveness for solving some planning problems.

Heuristic planning is a particular case of the concept of heuristic search used in problem solving. Planning domains are traditionally represented in the planning literature by a set of STRIPS [11] or ADL [34] operators describing a set of available actions. Heuristic planning requires a representation of planning domains which includes as well heuristic information about the subregions of the search space in which a successful plan solution may be found. In heuristic search, this information is usually described by a heuristic function which estimates the distance of a given state to the nearest goal states in the search space. Heuristic planning approaches, such as [4,37,38], use different representation formalisms to express that information. These formalisms tend to be declarative [27] and make use of logical languages such as the *situation calculus* [32] or the language of *linear temporal logic*. Some of these approaches [4,37] focus on the pruning aspect of the heuristic information by detecting those states from which the search process should not proceed. Others, such as [22] and [38], allow the representation of information that can also be used for leading the search process in promising directions and for establishing a partial order among the states from which the search process can proceed.

Heuristic planning differs from the approach to *planning as heuristic search* described in [6] in the fact that the heuristic information is provided by the user, rather than automatically generated by the planner from the description of a planning problem. The relation between the domain dependent approach to planning, represented by *heuristic planning*, and the domain independent approach, represented by *planning as heuristic search*, is very interesting in itself and its study could bring new ideas for both areas of research. Unfortunately, it falls out of the scope of this paper.

The rest of the paper is organized as follows. Section 2 describes the approach to heuristic planning proposed in this paper, which is based on a declarative formalization of strategies for action selection proposed in [38]. Section 3 compares this approach with the proposals of Bacchus and Kabanza [4], Reiter [37] and Doherty and Kvarnstrom [9]. Section 4 presents the results of some experiments which compare the performance of the heuristic forward chaining planner proposed in this paper with that of the planners described in [4,9,37]. Section 5 shows how heuristic information can be effectively used in the context of answer set planning, and presents the results of some experiments which improve the scalability of current answer set planners. Finally, Section 6 summarizes our main conclusions.

2. Heuristic planning

We describe a heuristic planning system based on the proposals of [38] for the declarative formalization of strategies for action selection. The *heuristic forward chaining*

planner (HFCP) has been implemented in Prolog and uses the representation scheme for planning domains described in that paper.

A *planning domain* is specified by a set of available actions and a strategy for action selection. A *strategy for action selection* is a consistent set of action selection rules. An *action selection rule* [15] is an implication whose antecedent is a formula of the situation calculus [32], and whose consequent can take one of the following forms: $Good(a, s)$, $Bad(a, s)$ or $Better(a, b, s)$. The intuitive interpretation of these predicates is that performing action a at situation s is *good*, *bad* or *better* than performing action b.

The heuristic forward chaining planner HFCP explores the space of situations that can be reached from the initial situation performing executable sequences of actions using a heuristic search strategy, which is guided by the strategy for action selection supplied by the user as follows.

The predicate *Good* is used to characterize *optimal actions*. These are actions whose execution always leads to an optimal plan solution.[1] This can be observed by examining the heuristics for the blocks world described below. Actions of the type *good* are used then to direct the search in the right direction, because they guarantee that an optimal solution will be found in that direction. They are used, therefore, to restrict the search to the subtree generated by applying a good action to the current situation.

The predicate *Bad* is used to characterize those actions which cannot be part of an optimal plan solution. Therefore, every subplan containing a bad action should not be considered by the planning system. Actions of the type *bad* are used, then, to prune the search tree considered by the planner at each situation. If an action is bad for the current situation, then the planner knows that it needs not generate or consider the situation resulting from applying that action to the current situation, nor any of the descendants of that situation. They are used, therefore, to avoid the generation of subtrees which cannot contain an optimal plan solution.

The predicate *better* is used to establish a partial order among actions which are not known to be good or bad for a particular situation. In some aspects, it acts as a heuristic function in a heuristic search algorithm, because it can be used to order the set of possible actions in the current situation. The predicate better is used, then, to control the backtracking process of the search strategy, determining at each situation the action that should be tried out of the remaining untried actions. The possibility of controlling the backtracking process by means of action selection rules of the form $FB \rightarrow Better(a, b, s)$ distinguishes the heuristic forward chaining planner described in this paper from the planner proposed in [39]. Lin [22] formalizes a partial order relation on actions as well, and uses it to provide a formal semantics for the Prolog *cut* operator.

In summary, the action selection mechanism of the heuristic forward chaining planner described in this paper works as follows: (1) If there is a good action for the current situation, it explores only the subtree of situations that can be generated from the situation resulting from applying this action to the current situation. (2) If there is no good action for the current situation, it explores the subtrees of situations that can be generated by applying

[1] In this paper, we assume that an optimal plan solution is a plan with a minimum number of actions. This notion of optimality is different from those proposed in [23].

nonbad actions to the current situation using the partial order established by the predicate *better* to guide its search process. A formal model of the action selection mechanism of the heuristic forward chaining planner described in this section is presented in Appendix A.

2.1. An example of strategy for action selection

We present an example of a strategy for action selection that can be used for planning in the blocks world. The strategy consists of a set of action selection rules (represented as Prolog rules) which describe some *heuristics* for moving blocks in order to solve planning problems in the blocks world.

The heuristics use three concepts which are not included in the usual descriptions of the blocks world. Therefore, their definitions must be part of the specification of the strategy for action selection. A block is in *final position* if it is on the table and it should be on the table in the goal configuration, or if it is on a block it should be on in the goal configuration and that block is in final position. A block is in *tower-deadlock position* if it is not in final position and it is above a block it should be above in the goal configuration. If a block is in tower-deadlock position, then it must be moved somewhere else (for example, to the table) before it can be put in final position. A block is in *next-final position* if it should be on the table, or if it should be on another block that it is currently clear and in final position. A block in *next-final position* needs not be clear in the current situation. It can only be put in its final position in the next situation if it is cleared in the current situation. The heuristics used for planning in the blocks world are as follows:[2]

- If a block can be moved to its final position, this should be done right away.
- If a block is in *tower-deadlock* position, then put it on the table.
- Suppose a block `B1` is not in its final position, and it is on a block `B2` that is in final position and should have a third block `B3` on it. Suppose a block `B4` is on a block `B5` that is not in its final position or should be clear in goal configuration. Then it is better to move `B1` than to move `B4`.
- Suppose a block `B1` is on a block `B2` that is in *next-final* position (i.e., if `B2` is cleared then it can be put immediately in final position). Suppose a block `B3` is on a block `B4` that is not in *next-final* position, then it is better to move `B1` than to move `B3`.
- If a block is in its final position, do not move it.
- If a block is neither on the table nor in its final position and cannot be moved to its final position, do not move it to any place different from the table.[3]

The constant `t` denotes the table of the blocks world. The predicate `ong(X,Y)` holds for a block `X` and a block `Y`, if `X` is on `Y` in the goal configuration of a planning problem. Similarly, `aboveg(X,Y)` holds if `X` is above `Y` in the goal configuration.

[2] The concept of *final position* and some of the heuristics described below were first proposed in [29].

[3] It is possible to obtain an optimal plan by moving such a block onto another block that is in final position and should be clear in the goal configuration. We have chosen not to consider those plans, because they are essentially equivalent to the plans that move such blocks to the table.

```
/* DEFINITIONS */

above(X,Y,S) :- on(X,Y,S).
above(X,Y,S) :- on(X,Z,S), above(Z,Y,S).

aboveg(X,Y) :- ong(X,Y).
aboveg(X,Y) :- ong(X,Z), aboveg(Z,Y).

final(X,S) :- ong(X,t), on(X,t,S).
final(X,S) :- ong(X,Y), not(Y=t), on(X,Y,S), final(Y,S).

tower_d(X,S) :- aboveg(X,Y), not(Y=t), above(X,Y,S),
                not(final(X,S)).

next_final(X,S) :- ong(X,Y), not(final(X,S)), (Y=t;
                   (clear(Y,S), final(Y,S))).

/* ACTION SELECTION RULES */

good(move(X,Y,Z),S) :- on(X,Y,S), clear(X,S), ong(X,Z),
     not(final(X,S)), (Z=t ; (clear(Z,S), final(Z,S))).
good(move(X,Y,t),S) :- on(X,Y,S), clear(X,S),
                       tower_d(X,S).

better(move(_,Y,t),move(_,W,t),S) :- final(Y,S),
       ong(_,Y), (not(ong(_,W)) ; not(final(W,S))).
better(move(_,Y,t),move(_,V,t),S) :- next_final(Y,S),
       not(next_final(V,S)).

bad(move(X,_,_),S) :- final(X,S).
bad(move(X,_,Z),S) :- ong(X,W), not(Z=t),
   (not(clear(W,S)); not(final(W,S))).
```

3. Pruning approaches

3.1. TLPlan

In [4] and [5], a planning system called TLPlan, which uses first order linear temporal logic to represent search control knowledge, is described. This logic is interpreted over sequences of worlds. In particular, the *goal modality* and *temporal modalities* ($\bigcup$ until, $\Box$ always, $\Diamond$ eventually and $\bigcirc$ next) are used to assert properties of world sequences. A *search control formula* describing the search control strategy to be used by the planner is specified in this logic. This formula describes properties the sequences of worlds generated by applying acceptable plans to the initial situation should satisfy.

The planner uses a *progression algorithm* which serves as the basis for an incremental mechanism that allows checking whether a plan prefix, generated by forward chaining, can lead to a plan that satisfies the search control formula. Whenever a plan prefix does not satisfy the search control formula, all the sequences of actions starting with that prefix are pruned from the search tree of executable sequences of actions. This mechanism allows reducing the search space by avoiding the exploration of sequences of actions which can never be extended to acceptable plans solutions.

Interesting experiments in which TLPlan performs better than state of the art planners, such as Blackbox [16] and IPP [17], in various test domains using search control formulas are described in [5]. Blackbox and IPP are both state of the art planning systems. They were the best performers in the AIPS'98 planning competition [1]. TLPlan did not compete however in AIPS'00 [2]. The best performer in the domain dependent track of the planning competition AIPS'00 was TALPlan [18]. TALPlan's approach for the representation and use of search control knowledge is similar to that of TLPlan. We describe some specific features of TALPlan at the end of this section.

TLPlan is an interesting example of a heuristic forward chaining planner in which search control knowledge is expressed in terms of properties the sequences of worlds generated by *acceptable plans* must satisfy. This heuristic information is used for pruning the search tree during the search process, but it is not exploited for directing the search in promising directions or for establishing a partial order on the set of plan prefixes which may be extended to acceptable plans solutions. The search control formula used for the blocks world in [5] is described below.

$$goodtower(x) \equiv clear(x) \wedge \neg Goal(holding(x)) \wedge goodtowerbelow(x)$$

$$\begin{aligned} goodtowerbelow(x) \equiv\ & (ontable(x) \wedge \neg\exists[y : Goal(on(x, y))]) \vee \exists[y : on(x, y)] \\ & \neg Goal(ontable(x)) \wedge \neg Goal(holding(y)) \wedge \neg Goal(clear(y)) \wedge \\ & \forall[z : Goal(on(x, z))]z = y \wedge \forall[z : Goal(on(z, y))]z = x \wedge goodtowerbelow(y) \end{aligned}$$

$$badtower(x) \equiv clear(x) \wedge \neg goodtower(x)$$

$$\begin{aligned} \Box\ ((&\forall[x : clear(x)]goodtower(x) \rightarrow \bigcirc(clear(x) \vee \\ & \exists[y : on(y, x)]goodtower(y))) \wedge (badtower(x) \rightarrow \bigcirc\neg\exists[y : on(y, x)]) \wedge \\ & (ontable(x) \wedge \exists[y : Goal(on(x, y))]\neg goodtower(y) \rightarrow \bigcirc\neg holding(x))) \end{aligned}$$

It is easy to observe that the concepts defined by the predicates $goodtowerbelow(x)$ and $final(x, s)$ (used in the strategy for action selection of Section 2.1) are equivalent, although the definition of $final(x, s)$ is considerably simpler. The search control formula described above rules out any sequence of actions that disassembles a good tower, places a block on top of a bad tower, or picks up a block on the table unless that block can be moved to its final position. The two action selection rules which define the predicate $bad(a, s)$ in the strategy for action selection of Section 2.1 prune the same sequences of actions as this search control formula. Because, they forbid actions which move blocks that are already in final position (i.e., which disassemble good towers), and actions which move blocks that cannot be moved to their final positions to places different from the table.

The search control formula used by TLPlan does not recognize however some other sequences of actions that cannot be extended to construct optimal plans for the blocks world. These correspond to the action selection rules which characterize good actions in the strategy for action selection described in Section 2.1. In order to take these heuristics into account in the representation formalism used by TLPlan, it would be necessary to rule out any sequence of actions which does not include a good action when that action can be performed in the current state. However, this approach would not direct the search in the right direction, as the predicate good does. TLPlan would have to generate each successor sequence of actions, test it to determine whether it falsifies the search control formula, and prune the subtree of sequences of actions that can be generated from it in that case. Finally, it should be noted that TLPlan has also been applied to generate plans that satisfy *temporally extended goals* [3].

TALPlan [9] is a forward chaining planner which uses domain dependent knowledge to control search in the state space generated by action invocation. The domain dependent control knowledge, background knowledge, plans and goals are all represented using formulas in a temporal logic called TAL. TAL has been developed independently as a formalism for specifying agent narratives and reasoning about them. The domain control knowledge used by TALPlan in the blocks world is a translation of the search control formula used by TLPlan to the temporal logic TAL. Search control formulas are used for pruning the search space in a similar way in both systems, with the exception that TLPlan uses formula progression and the efficient version of TALPlan uses formula evaluation combined with some optimization techniques. These techniques allow moving part of the search control information from the search control formula to the preconditions of the operators. This improves performance significantly, since control rule violations can be detected before the planner even attempts to invoke an operator. It should be observed that this form of *precondition control* information is equivalent to the information encoded in action selection rules of the form $F(s) \rightarrow Bad(a, s)$. These rules are evaluated before operator invocation as well, and are therefore as efficient as precondition control information. Finally, and in addition to its impressive performance, TALplanner has the advantage that it has been extended to deal with concurrent actions, actions with time dependent effects, and allocation of resources [18].

3.2. The predicate badSituation

In [37] a forward chaining planner which uses a regression based theorem prover and an iterative deepening search strategy is proposed. The planner requires the following types of information from the user: (1) a predicate $goal(s)$, which is true if situation s satisfies the conditions of the goal for which a plan is sought; (2) a set of *action precondition and successor state axioms* [36] for the primitive actions of the domain; and (3) a predicate $badSituation(s)$, which is true if situation s is considered to be a bad situation for the planner to explore. The planner is implemented in GOLOG [19], and it has been extended to deal with incomplete initial situations [12].

The representation formalism used by the heuristic forward chaining planner described in Section 2 is more expressive than that used in [37], in the sense that it allows the representation of *positive heuristics* (the predicate *good* tells the planner what to do),

and heuristics that establish preferences among actions (the predicate *better* establishes a partial order among actions). The predicate *badSituation*(s) allows pruning the search space by characterizing those situations from which a successful plan cannot be reached, but it does not allow guiding the search in promising directions or controlling the backtracking process establishing a partial order on successor situations, as the predicates *good* and *better* do.

The heuristics used in [37] for the blocks world are described below. They are Prolog rules which define the predicate *badSituation*(s), which is used for pruning the search tree of reachable situations explored by the planner.

```
badSituation(do(move(X,Y),S)) :-
 not goodTower(X,do(move(X,Y),S)).

badSituation(do(moveToTable(X),S)) :- goodTower(X,S).

badSituation(do(moveToTable(X),S)) :-
 not goodTower(X,do(moveToTable(X),S)),
 existsActionThatCreatesGoodTower(S).

existsActionThatCreatesGoodTower(S) :- (A=move(Y,X);
 A=moveToTable(Y)), poss(A,S), goodTower(Y,do(A,S)).
```

These heuristics are very similar to the heuristics used by TLPlan. They characterize those situations from which a successful plan cannot be reached. However, as situations are equivalent to sequences of actions in the axiomatization of the situation calculus used by [37], one can safely say that the heuristics characterize, in fact, bad sequences of actions as the heuristics used by TLPlan do. The main difference is that TLPlan uses linear temporal logic formulas to characterize legal sequences of actions, pruning away those sequences of actions which do not satisfy the search control formula; and [37] uses situation calculus formulas to characterize bad sequences of actions, pruning away those sequences of actions that satisfy the predicate *badSituation*.

The heuristics used in [37] prune however more situations than those used by TLPlan, because in addition to forbidding building up bad towers and disassembling good towers they also recognize the fact that whenever there is an action that builds a good tower (i.e., a good action) the situation resulting from performing any action which does not build a good tower should be considered as a bad situation. This amounts to forbidding any nongood actions whenever there is a good action for the current situation, as the mechanism for action selection of the heuristic forward chaining planner described in section 2 does. This domain independent heuristic could be incorporated into the planner described in [37] by adding the following rule to the definition of the predicate *badSituation*.

```
badSituation(do(A,S)) :- poss(B,S), good(B,S),
                         not(good(A,S)).
```

The addition of this rule would allow the planner to use action selection rules of the form $FG \rightarrow Good(a, s)$ for pruning its search space, but it would not allow it to use them for directing its search process as the heuristic forward chaining planner described in Section 2 does.

The heuristics in [37] do not consider however an important concept which allows guiding the search process as well. This is the concept of a block being in *tower-deadlock position*. Therefore, they cannot discriminate between actions that move arbitrary blocks to the table (which are not necessarily optimal and can be postponed), and actions that move blocks in tower deadlock position to the table (which are necessary and should be executed right away). This is the meaning of the second action selection rule in the definition of the predicate *good* in the action selection strategy of Section 2.1.

4. Empirical results

We have done some experiments in order to compare the performance of the heuristic forward chaining planner (HFCP) described in Section 2 with respect to state of the art planning systems such as TLPlan [5], TALPlan [9] and R [25]. System R is a planner based on a regression/progression algorithm which uses domain specific information to order subgoals, prune unachievable goals, and determine the way a subgoal is solved by regressing it to a new conjunctive subgoal.

First, we have applied HFCP and TLPlan to solve some blocks world problems. HFCP used the strategy for action selection described in Section 2.1, and TLPlan the search control formula described in Section 3.1. The first problem set (shown in Table 1) consists of 10 randomly generated blocks world problems of 25 blocks. The second problem set (shown in Table 2) consists of 6 blocks world problems of different sizes. The sizes of the problems are specified in the first column of Table 2. For each problem, we have computed the number of blocks that are initially in final and tower deadlock positions (columns *Final* and *TD*). The numbers in the columns *Steps* and *Time* correspond to the number of steps of the plans found by the each planner and the time in milliseconds spent on planning.

In order to make a fair comparison we have divided by two the number of steps of the plans found by TLPlan. This is due to the fact that the action $move(x, y, z)$, used in the action selection strategy of Section 2.1, corresponds to two actions of the form $pickup(x)$, $putdown(x)$, $stack(x, y)$ or $unstack(x, y)$ used by TLPlan. Therefore, the plans obtained by TLPlan should be twice as long as the plans obtained by the heuristic forward chaining planner.

Comparing the numbers in the columns *StepsHFCP* and *StepsTLP*, it can be observed that TLPlan cannot find optimal plans (i.e., plans with a minimum number of steps) for 10 of the 16 problems posed. The heuristic forward chaining planner obtains optimal plans for the 16 problems. As far as planning time is concerned, the heuristic forward chaining planner is faster than TLPlan.

We have applied HFCP to solve a number of blocks world problems from the Artificial Intelligence Planning & Scheduling competition [2]. These are problems of 100 to 500 blocks. Tables 3 and 4 describe our results and compare them to the results reported for TALPlan and system R in AIPS 2000. We have used two different versions of the heuristic

Table 1
Problems of 25 blocks

Prob	Final	TD	Steps HFCP	Time HFCP	Steps TLP	Time TLP
25-1	1	2	26	0	26	58
25-2	0	11	36	20	38	91
25-3	3	1	23	0	25	58
25-4	7	0	18	10	20	52
25-5	7	2	20	10	20	46
25-6	1	4	28	10	30	68
25-7	1	6	30	20	37	91
25-8	1	13	37	20	37	85
25-9	1	3	27	20	29	68
25-10	1	7	31	20	32	84

Table 2
Problems of different sizes

Size	Final	TD	Steps HFCP	Time HFCP	Steps TLP	Time TLP
5	2	0	4	0	5	4
13	1	3	15	0	15	19
15	2	0	14	0	18	26
19	2	0	18	0	25	47
25	5	1	22	10	22	51
50	24	0	26	0	26	158

forward chaining planner. HFC1 refers to a previous version of the planner described in [39, 40] which uses strategies for action selection for directing and pruning the search process, but it does not use them for controlling the backtracking process as the planner described in Section 2 does. The strategy for action selection used by HFC1 is equivalent to the action selection rules that define the predicates *good* and *bad* in the action selection strategy of Section 2.1. HFC2 refers to the heuristic forward chaining planner described in Section 2, and uses the strategy for action selection described in Section 2.1. It should be observed as well that the results for TALPlan and R were generated on a 500 MHz Pentium III machine with 1 GB of RAM memory, whereas the results for HFC1 and HFC2 were generated on a 733 MHz Pentium III machine with 128 MB of RAM memory.

Table 3
Planning time: problems from AIPS 2000

Size	TAL	R	HFC2	HFC1	R – HFC1
100-0	0.31	2.49	1.27	0.69	1.8
100-1	0.31	2.48	1.88	0.89	1.59
200-0	0.48	9.73	36.32	8.83	0.9
200-1	0.48	9.68	22.13	9.6	0.08
250-0	0.58	15.34	12.52	8.26	7.08
250-1	0.61	15.51	29.41	12.31	3.2
300-0	0.83	22.71	6.52	6.52	16.19
300-1	0.70	22.99	38.62	19.86	3.13
350-0	0.84	31.64	59.84	59.89	−28.25
350-1	0.82	31.03	22.98	23.02	8.01
400-0	1.12	39.28	64.4	21.83	17.45
400-1	1.00	41.26	175.85	47.72	−6.46
425-0	1.14	45.17	158.01	46.56	−1.39
425-1	1.24	46.87	76.12	38.39	8.48
450-0	1.45	53.63	58.64	28.35	25.28
450-1	1.23	50.45	41.17	20.06	30.39
475-0	1.39	59.45	223.05	61.86	−2.41
475-1	1.47	58.83	216.37	43.57	15.26
500-0	1.57	65.35	67.88	29.99	35.36
500-1	1.41	64.01	121.56	42.01	22

Table 3 shows the time spent on planning by each system (times are given in seconds). TALPlan is clearly faster than the other systems. The first version of the heuristic forward chaining planner is on average slightly faster than system R. Column $R - HFC1$ shows the difference between the time spent on planning by system R and HFC1. HFC2 is slower than the other planners, but generates plans of better quality as it can be observed in Table 4.

Table 4 compares the performance of the planners in terms of plan quality. HFC1 and HFC2 generate shorter plans than system R for all problems except problem 300-1. HFC2 generates shorter plans than TALPlan for all the problems, and HFC1 for all problems except problem 425-1. Columns T − H2 and R − H2 show the difference on number of steps between the plans generated by TALPlan and HFC2, and system R and HFC2, respectively. In both cases, we have multiplied by two the number of steps of the plans generated by HFC2 before computing the difference.

Finally, we have applied HFCP to solve some problems from the planning domain of *logistics*. The problems in this domain consist of a set of objects (packages) which are initially located at different places in various cities and must be transported to their final destinations. There are two types of vehicles that can be used for this purpose: trucks and airplanes. Trucks can be used to transport objects within a city, and airplanes to transport them between two airports.

Table 5 compares the performance of TALPlan, R and HFC1 on a number of logistics problems used in the second track of the planning competition [2]. The sizes of the problems are indicated in the first column, and range from 96 to 100 objects. The performance of the planners is compared in terms of the time spent on planning (expressed in seconds) and the length (i.e., number of the steps) of the plans generated for each for each

Table 4
Plan length: problems from AIPS 2000

Size	TAL	HFC2	HFC1	R	T – H2	R – H2
100-0	372	183	183	368	6	2
100-1	370	185	185	374	0	4
200-0	736	357	363	732	22	18
200-1	744	355	360	734	34	24
250-0	946	465	465	942	16	12
250-1	956	473	473	950	10	4
300-0	1174	577	577	1158	20	4
300-1	1158	573	573	1138	12	−8
350-0	1340	663	663	1330	14	4
350-1	1324	659	659	1342	6	24
400-0	1556	774	774	1562	8	14
400-1	1540	753	754	1516	34	10
425-0	1620	801	804	1646	18	44
425-1	1622	808	815	1646	6	30
450-0	1770	876	876	1754	18	2
450-1	1740	865	865	1752	10	22
475-0	1840	916	916	1838	8	6
475-1	1828	906	906	1844	16	32
500-0	1948	965	965	1962	18	32
500-1	1954	972	972	1964	10	20

Table 5
Logistics: problems from AIPS 2000

Size	Time TAL	Time R	Time HFC1	Steps TAL	Steps R	Steps HFC1
96-0	0.794	505.99	12.470	627	3132	140
96-1	0.554	503.5	10.070	616	3082	129
97-0	0.544	560.31	12.860	605	3371	134
97-1	0.544	579.86	12.610	601	3369	138
98-0	0.574	548.21	9.570	630	3391	127
98-1	0.534	542.88	9.820	576	3273	124
99-0	0.734	539.21	15.530	631	3361	148
99-1	0.564	574.41	12.980	623	3432	139
100-0	0.644	640.43	16.220	646	3540	144
100-1	0.594	633.67	16.460	638	3569	147

problem.[4] It can be observed that HFC1 generates shorter plans than TALPlan and system R for all the problems, it is slower than TALPlan, but it is much faster than system R. The action selection strategy and problem formulation used by HFC1 for the domain of logistics is described in Appendix B.

[4] The numbers in the columns *Time TAL*, *Time R*, *Steps TAL* and *Steps R* correspond to the results reported for these systems in [2].

5. Heuristic answer set planning

In this section, we describe some experiments on the generation of *concurrent plans* using domain dependent heuristics in the context of *answer set planning* [20,43]. The concept of *answer set* [13,14] was originally defined to provide a declarative semantics for negation as failure as implemented in existing Prolog systems. *Answer set programming* [26,33] is based on the idea of representing a computational problem by a logic program whose answer sets correspond to solutions, and using an answer set solver to find the answer sets for that program. In this paper, we use an answer set solver called SMODELS [42]. This system computes answer sets for finite programs without disjunction or negation as failure in the heads of the rules. Other systems capable of computing answer sets are DLV, DERES and CCALC [45].

The key idea of *answer set planning* [43] consists in representing planning domains, such as the blocks world, in the form of history programs. A *history program* is a logic program whose answer sets represent possible histories or evolutions of the system over a fixed time interval. An important advantage of answer set planning is that the representation of properties of actions is easier when logic programs are used instead of classical logic, because of the nonmonotonic character of negation as failure [20]. Results of computational experiments that use SMODELS for planning are reported in [8,10,33].

In [21] a history program which can be used for planning in the blocks world is described in detail. We summarize below its main components and compare it to the SMODELS program we propose for doing heuristic answer set planning in the blocks world. The answer sets of this program represent possible evolutions of the blocks world over the time interval `0`, ..., `lasttime`, for a fixed positive integer `lasttime`. A history of the blocks world is characterized by the truth values of atoms of two kinds: `on(B,L,T)` (block `B` is on location `L` at time `T`) and `move(B,L,T)` (block `B` is moved to location `L` between times `T` and `T+1`).

When a history program is available, we can find a plan of length `lasttime` that solves a given planning problem or establish that such a plan does not exist by extending the history program with constraints representing the initial and goal states of the problem. The answer sets for the extended program correspond to the plans of length `lasttime` that lead from the initial state to the goal state. A planner would invoke a system for computing answer sets to find an answer set X for the extended program, and then return the list of atoms in X that represent actions.

The SMODELS program described in [21] (shown below) consists of three main components: (1) a *generate part*, which generates potential solutions; (2) a *define part*, which derives logical consequences from the specification of a potential solution; and (3) a *test part*, which checks whether a potential solution corresponds to an executable plan and satisfies the goal conditions. In particular, the GENERATE section of the program below defines a potential solution to be an arbitrary set of *move* actions executed prior to `lasttime`. The DEFINE section describes the sequence of states corresponding to the execution of a given plan. Each sequence of states is represented by a complete set of literals of the form `on(B,L,T)`. The TEST part prohibits the execution of actions that would create physically impossible configurations of blocks, such as moving two

blocks onto the same block, and checks whether the goal configuration is achieved by the plan.

```
time(0..lasttime).

location(B) :- block(B).
location(table).

% GENERATE

{move(B,L,T) : block(B) : location(L)} :- time(T),
                                          T<lasttime.

% DEFINE

% effect of moving a block
on(B,L,T+1) :- move(B,L,T), block(B), location(L),
               time(T), T<lasttime.

% inertia
on(B,L,T+1) :- on(B,L,T), not on'(B,L,T+1), block(B),
               location(L), time(T), T<lasttime.

% uniqueness of location
on'(B,L1,T) :- on(B,L,T), L!=L1, block(B),
               location(L), location(L1),
               time(T).

% TEST

% on' is the negation of on
:- on(B,L,T), on'(B,L,T), block(B), location(L),
   time(T).

% two blocks cannot be on top of the same block
:- 2 {on(B1,B,T) : block(B1)}, block(B), time(T).

% a block can't be moved unless it is clear
:- move(B,L,T), on(B1,B,T), block(B), block(B1),
   location(L), time(T), T<lasttime.

% a block can't be moved onto a block that is being
  moved
:- move(B,B1,T), move(B1,L,T), block(B), block(B1),
  location(L), time(T), T<lasttime.
```

```
% PLANNING PROBLEM

const lasttime=3.

% initial configuration
on(1,2,0). on(2,table,0). on(3,4,0). on(4,table,0).
on(5,6,0). on(6,table,0).

% goal test
:- not on(3,2,lasttime). :- not on(2,1,lasttime).
:- not on(1,table,lasttime). :- not on(6,5,lasttime).
:- not on(5,4,lasttime). :- not on(4,table,lasttime).
```

In this paper, we take a slightly different approach to answer set planning. The first difference consists in using the answer set solver to *construct plans incrementally*. Instead of generating a complete plan solution in a single call to the answer set solver, we call it a number of times in order to construct a plan incrementally. First, we check whether the initial configuration satisfies the goal condition. If it does, the empty plan is returned as a solution. Otherwise, we compute the set of actions that should be executed concurrently in the current configuration, construct the configuration resulting from executing them, and apply the goal test to the resulting configuration. This process continues until a configuration is reached where the goal test is satisfied, or until the length of the current plan is greater than the depth limit of the search strategy. Plans are constructed thus incrementally, using a forward chaining approach which requires the answer set solver to construct only time histories of length one.

Table 6 shows the results of some experiments in which we solve ten blocks world problems of 25 blocks in an average time of 22 seconds, and ten blocks world problems of different sizes. The experiments have been carried out using SMODELS 2.26 [41]. For each problem we have computed the *time* spent on planning (times are in *s* seconds, *m* minutes, or *h* hours depending on the amount), the total number of actions included in

Table 6
Problems of 25 blocks (left) and different sizes (right)

Prob	Time	Actions	Length	Size	Time	Actions	Length
25-1	25.39 s	26	26	15	10.19 s	18	10
25-2	32.77 s	39	33	19	11.53 s	25	11
25-3	15.38 s	26	15	50	2.58 s	26	2
25-4	4.36 s	20	4	75	1.79 m	111	79
25-5	9.31 s	20	9	100-0	3.24 m	188	125
25-6	20.27 s	29	20	100-1	3.69 m	187	144
25-7	32.6 s	36	32	200-0	15.35 m	371	224
25-8	34.15 s	38	34	200-1	15.54 m	373	227
25-9	22.09 s	30	22	300-0	1.82 h	588	570
25-10	24.88 s	33	25	300-1	1.7 h	581	542

each plan (column *actions*), and the *length* (number of time steps) of the solution returned by the planner. The problems of 15 and 19 blocks correspond to the problems *bw-large.c* and *bw-large.e* in [8,33], respectively. In [8], the best time obtained for *bw-large.c* is 190 seconds and for *bw-large.e* 1933 seconds. The times reported in [33] for these problems are 25 seconds and 100 seconds, respectively. These are execution times of the SMODELS module when given a parsed and grounded program as input. The times reported in Table 6 and those from [8] include the time spent on parsing and grounding.

The second difference between our approach to answer set planning and previous ones is the use of *heuristic information in history programs*. The following history program for the blocks world has been used in the experiments described in Table 6.

First, we show the description of a planning problem. The predicate `on(B,L,T)` from the previous program is replaced by three predicates: `on0(B,L)`, for the initial configuration, or current configuration during the search process; `ong(B,L)`, for the goal configuration; and `on1(B,L)`, for the successor configuration during the search process. The advantage of using these predicates is that the predicates `on0(B,L)` and `ong(B,L)` are domain predicates, because of our complete knowledge about the initial and goal configurations, and therefore need not be computed during the answer set construction process.

```
% PLANNING PROBLEM
% initial configuration
on0(1,2). on0(2,table). on0(3,4). on0(4,table).
on0(5,6). on0(6,table).

% goal configuration
ong(3,2). ong(2,1). ong(1,table). ong(6,5).
ong(5,4). ong(4,table).
```

We introduce now some of the concepts used in the action selection strategy for the blocks world described in Section 2.1. The predicates associated with these concepts are domain predicates as well, because they are defined in terms of domain predicates only.

```
location(B) :- block(B).
location(table).

nclear(B) :- on0(B1,B), block(B), block(B1).

clear(B) :- not nclear(B), block(B).

final(B) :- on0(B,table), ong(B,table), block(B).
final(B) :- on0(B,B1), ong(B,B1), final(B1),
            block(B), block(B1).

navailable(B) :- not final(B), block(B).
```

```
navailable(B) :- nclear(B), block(B).

above(B,B1) :- on0(B,B1), block(B1), block(B).
above(B,B1) :- on0(B,B2), above(B2,B1), block(B),
               block(B1), block(B2).

aboveg(B,B1) :- ong(B,B1), block(B1), block(B).
aboveg(B,B1) :- ong(B,B2), aboveg(B2,B1), block(B),
                block(B1), block(B2).

towerD(B) :- above(B,B1), aboveg(B,B1), not final(B),
             block(B), block(B1).
```

We describe below the fragment of our program that corresponds to the *generate* part of the blocks world program presented in [21]. Instead of using a single rule that generates all subsets of atoms of the form `move(B,L,T)`, we use two types of rules which implement *action selection rules* of the form $F(a,s) \rightarrow \mathit{Good}(a,s)$ and $G(a,s) \rightarrow \mathit{Bad}(a,s)$ in the input language of SMODELS.

As we have explained in Section 2, we use the predicate *Good(a,s)* to characterize optimal actions. These actions can be safely included in a plan, because their execution always leads to an optimal solution. In terms of SMODELS, this is equivalent to saying that an atom of the form `move(B,L)` must be in every answer set if that atom corresponds to an action that is good for the current configuration. The following rules implement that intuition: whenever the conditions of the antecedent of an action selection rule of the form $F(a,s) \rightarrow \mathit{Good}(a,s)$ hold in the current configuration, the atom corresponding to action a must be included in every answer set. The first two rules below implement the heuristic: if a block can be moved to final position, this should be done right away. The third rule corresponds to the heuristic: if a block is in tower deadlock position, put it on the table.

```
move(B,B1) :- clear(B), ong(B,B1), not on0(B,B1),
              final(B1), clear(B1), block(B), block(B1).

move(B,table) :- clear(B), ong(B,table),
                 not on0(B,table), block(B).

move(B,table) :- clear(B), towerD(B), block(B).
```

The rule described below allows the generation of actions that are executable and nonbad for the current configuration. In particular, it generates moves of the form `move(B,table)` for those blocks which are clear, not in final position, and cannot be moved directly to their final positions. The antecedent of this rule avoids the generation of bad moves, and therefore implements the same control mechanism as action selection rules of the form $G(a,s) \rightarrow \mathit{Bad}(a,s)$ described in Section 2. Finally, the last three rules simply force the execution of at least one action at each time step, in order to avoid the generation of plan steps without any action.

```
{move(B,table)}1 :- clear(B), ong(B,B1),
                    not on0(B,table), not towerD(B),
                    not final(B), navailable(B1),
                    block(B), block(B1).

% At least one move is required at each time step.
moving :- clear(B), move(B,table), block(B).
moving :- clear(B), ong(B,B1), final(B1), move(B,B1),
          block(B), block(B1).
:- not moving.
```

In general, it is possible that two actions are optimal when executed separately, but they are not when executed concurrently. This is not the case in our example, because moving a block to final position or to the table never interferes with any other subgoal. If the table could only accommodate a finite number of blocks, we would have to deal with that problem. In such a case, it is not a good idea to use rules that force the inclusion of good actions in the answer sets. It is better to allow the generation of nonbad actions and use the maximization capabilities of SMODELS to prefer answer sets that maximize the occurrence of compatible good actions in the answer set returned as a solution.

We describe now the rules we have used to implement the parts *define* and *test* of the history program presented in [21]. The rule for describing *the effect of moving a block* is replaced by the following two rules. The rule used in [21] generates $t(n+1)$ grounded clauses for each block in the problem, where n is the number of blocks and t is the value of the constant `lasttime`. The rules below generate at most two grounded clauses for each block that is clear in the current configuration. They take advantage of heuristic knowledge in order to recognize that only clear blocks can be moved, and that the heuristics for action selection only allow moving them to the table, or to the block they should be on, if that block is already clear and in final position. The effect of these rules is therefore simplifying the grounded theory resulting from applying `lparse` [44] to the history program. The number of clauses is reduced from $n(n+1)t$ to $2c$, where c is the number of clear blocks in the current configuration.

```
on1(B,B1) :- clear(B), ong(B,B1), final(B1), clear(B1),
             move(B,B1), block(B), block(B1).

on1(B,table) :- clear(B), move(B,table), block(B).
```

The rule for describing *inertia* in the program presented in [21] is very elegant and simple, but generates $n(n+1)t$ grounded clauses. We replace it by three rules that take advantage of domain dependent heuristics in order to reduce the number of grounded clauses generated to n. The heuristics used are: (1) if a block is not clear, its position does not change, because it cannot be moved; (2) if a block is in final position its position does not change, because the heuristics for action selection do not allow moving it; (3) if a block is clear and it is not moved to the table or to the block it should be on in the goal

configuration its position does not change, because the heuristics for action selection do not allow moving it anywhere else.

```
on1(B,L) :- on0(B,L), nclear(B), block(B), location(L).

on1(B,L) :- on0(B,L), clear(B), final(B), block(B),
            location(L).

on1(B,L) :- on0(B,L), clear(B), not final(B), ong(B,B1),
            navailable(B1), not move(B,table),
            not move(B,B1),
            location(L), block(B), block(B1).
```

The rest of the rules that appear in the *define* or *test* parts of the program presented in [21] are not needed, except for the rule that prohibits moving a block onto another block that is being moved at the same time. The rule used in that program is replaced by two rules again in order to reduce the number of clauses generated by the grounding process from $n^2(n+1)t$ to $2c$, where n is the number of blocks, t the value of the constant `lasttime`, and c the number of blocks which are clear in the current configuration.

```
:- ong(B,B1), clear(B), move(B,B1), move(B1,table),
   block(B1), block(B).
:- ong(B,B1), ong(B1,B2), clear(B), move(B,B1),
   move(B1,B2), block(B1), block(B), block(B2).
```

6. Conclusions

A major drawback of current logical reasoning systems, pointed out by John McCarthy in [30,31], is their inability to use domain and problem dependent heuristic advice to guide their search for solutions. In this paper, we have addressed this problem by presenting a scheme for the declarative formalization of heuristics and showing how a general purpose forward chaining planner can use declarative formalizations of heuristics to improve its performance in several domains.

In particular, we have introduced the notion of heuristic planning, and described a particular approach to heuristic planning based on a declarative formalization of strategies for action selection proposed in [38]. This approach has been compared with the proposals of Bacchus and Kabanza [4], Doherty and Kvarnstrom [9] and Reiter [37]. The heuristic information and declarative formalisms for the representation of heuristic knowledge used by these systems have been analyzed and compared in terms of their capacity of controlling the search process and their effectiveness for solving some planning problems. The following are some conclusions that can be drawn from this comparative analysis.

(1) The mechanism for action selection and the declarative formalization of strategies for action selection used by the heuristic planner described in this paper allow controlling

the search process in more sophisticated ways than the pruning approaches used by TLPlan [4] and the planning system described in [37]. In particular, they allow directing the search in promising directions, and controlling the backtracking process in such a way that actions are tried in order of their quality, which is established by the predicate *better*.

(2) The quality of the heuristic information used by the heuristic planner for the domain of the blocks world is slightly better than that used by TLPlan or the planning system described in [37]. In particular, it uses the concept of *tower deadlock* position, which is not considered in the heuristics used by the other systems.

(3) The regression theorem prover used by the planner described in [37] can be applied to the resolution of planning problems with incomplete initial situations [12], whereas the state update approaches used by TLPlan and the heuristic planner described in this paper cannot be applied to solve such problems.

(4) The representation formalism used by TLPlan has been applied to generate plans that satisfy *temporally extended goals* [3].

(5) TALplanner [9] is faster than the rest of the heuristic planners considered in this paper and it has been extended to deal with concurrent actions, actions with time dependent effects, and allocation of resources.

We have described the results of some experiments which show that the heuristic forward chaining planner proposed in Section 2 improves the performance of TLPlan in the domain of the blocks world. The improvement on performance allows constructing plans of better quality without sacrificing the amount of search required or the time spent on planning. The experiments have shown as well that the heuristic forward chaining planner generates shorter plans than TALPlan and system R for most problems in different domains, and that it is slower than TALPlan but faster than system R.

An additional advantage of the heuristic planner proposed in this paper is the availability of a formal model for its mechanism for action selection based on the situation calculus and predicate completion [7] (see Appendix A for a detailed description of the formal model). This formal model allows interesting forms of meta reasoning about declarative formalizations of strategies for action selection such as: (1) determining the correctness of a particular strategy; (2) updating and composing strategic knowledge from different sources; or (3) determining whether a set of action selection rules improve, are inconsistent or redundant with a particular strategy for action selection.

Finally, we have shown how heuristic information can be effectively used in the context of answer set planning, and presented the results of some experiments which may offer a potential solution to the scalability problems exhibited by current answer set planners.

Acknowledgements

The author would like to thank John McCarthy for interesting her on the declarative formalization of heuristics, and providing the intellectual and financial support necessary for the realization of this work. Vladimir Lifschitz contributed with valuable comments to the formalization and answer set planning sections of the paper. Ernest Davis and

the members of the Stanford Formal Reasoning Group, Eyal Amir and Aarati Parmar, contributed as well with useful comments. This research is partially supported by the Spanish Ministry of Science and Technology under project TIC 2000-0539.

Appendix A. Formal model

This section presents a formal model of the heuristic forward chaining planner described in this paper. The formal model is based on a formalization of search algorithms in the situation calculus proposed in [24]. The key idea in that paper is considering search algorithms as strict linear orders on sets of situations, and search pruning as a restriction of such ordering relations to a smaller set of situations. First, we define the set of situations which constitute the search space of a forward chaining planner. This is the set of situations that can be reached from the initial situation performing executable sequences of actions. It can be defined by the following foundational axioms [36].

$$\forall h, \vec{x}, \vec{y}\ \big(h(\vec{x}) = h(\vec{y}) \rightarrow \vec{x} = \vec{y}\big) \wedge \forall h, g, \vec{x}, \vec{y}\big(h(\vec{x}) \neq g(\vec{y})\big) \tag{A.1}$$

$$\forall s(\neg s < S_0) \wedge \forall s\big(s < \mathit{Result}(a, s_1) \leftrightarrow \mathit{Poss}(a, s_1) \wedge s \leqslant s_1\big) \tag{A.2}$$

$$P(S_0) \wedge \forall s, a\ \big(P(s) \wedge \mathit{Poss}(a, s) \rightarrow P\big(\mathit{Result}(a, s)\big)\big) \rightarrow \forall s P(s) \tag{A.3}$$

We assume uniqueness of names for every function symbol and every pair of distinct function symbols.[5] The constant symbol S_0 denotes the initial situation. The function *Result* maps an action a and a situation s into the situation resulting from performing action a at situation s. The predicate $\mathit{Poss}(a, s)$ is true provided action a can be executed at situation s. The expression $s < s_1$ means that s_1 can be *reached* from s performing a nonempty sequence of *executable* actions, $s \leqslant s_1$ is an abbreviation for $s < s_1 \vee s = s_1$. Finally, we introduce an axiom of induction for situations which allows proving that a property holds for all situations. This axiom constrains the domain of situations to those that can be reached from the initial situation performing executable sequences of actions.

In [24] the strict linear order associated with depth first search is described by the predicate $\mathit{DFS}(s_1, s_2)$, which is true provided situation s_1 is explored before situation s_2 by depth first search. This predicate is defined by the following axiom, which assumes the existence of a strict linear order $\mathit{Pref}(a, b)$ on actions.

$$\mathit{DFS}(s_1, s_2) \leftrightarrow s_1 < s_2 \vee \exists a, b, s\big(\mathit{Pref}(a, b) \wedge \mathit{Result}(a, s) \leqslant s_1 \wedge \mathit{Result}(b, s) \leqslant s_2\big) \tag{A.4}$$

The search algorithm used by the heuristic forward chaining planner described in Section 2 differs from standard depth first search in some aspects: (1) it uses limited depth first search, i.e., it explores situations of depth less than or equal to a given depth limit l; (2) it prunes the search space by considering only situations that are generated by applying sequences of *selectable* actions to the initial situation; and (3) it uses the predicate $\mathit{Better}(a, b, s)$, which depends on the current situation s, to determine the preference or

[5] The symbols h and g are meta-variables ranging over distinct function symbols; $\vec{x}$ and $\vec{y}$ denote tuples of variables.

strict linear order relation on actions used by Axiom (A.4). The first two aspects can be viewed as pruning mechanisms, whereas the third one can be seen as a redefinition of the preference relation on actions $Pref(a, b)$.

We define a predicate $Back(s, l)$ which is true for a situation s and a depth limit l if s is a backtracking node for the search algorithm. In limited depth first search, backtracking nodes are those situations generated by sequences of executable actions whose length is equal to the depth limit l given to the algorithm. Using this predicate, we define a predicate $Sel(s, l)$ which characterizes those situations that are not pruned away by the depth limit or the action selection mechanism of the heuristic forward chaining planner. These are the situations that can be generated from the initial situation by applying sequences of executable and selectable actions whose length is less than or equal to the depth limit l. The action selection mechanism of the heuristic forward chaining planner is described by Axiom (A.7): *an action a is selectable at situation s if a is executable and good for that situation; or if there are no actions which are executable and good for s, and a is executable and nonbad for that situation.*

$$Length(S_0) = 0 \wedge Length\big(Result(a, s)\big) = 1 + Length(s) \tag{A.5}$$

$$Back(s, l) \leftrightarrow Length(s) = l \tag{A.6}$$

$$Select(a, s) \leftrightarrow Poss(a, s) \wedge \big(Good(a, s) \vee \big(\neg Bad(a, s) \wedge \neg\exists b(Poss(b, s) \wedge Good(b, s))\big)\big) \tag{A.7}$$

$$Sel(s, l) \leftrightarrow s = S_0 \vee \exists a, s_1\big(Sel(s_1, l) \wedge \neg Back(s_1, l) \wedge Select(a, s_1) \wedge s = Result(a, s_1)\big) \tag{A.8}$$

The preference relation $Pref(a, b)$ is redefined as follows in order to control the backtracking process using the heuristic knowledge provided by the predicate $Better(a, b, s)$. Let $<_{alph}$ denote the alphabetic order, and $Better^*(a, b, s)$ the transitive closure of the predicate *Better*. The predicate $Pref(a, b, s)$ is true provided action a is tried before action b by the search algorithm when it explores the set of situations that can be generated from situation s.

$$Better^*(a, b, s) \leftrightarrow Better(a, b, s) \vee \exists c\big(Better(a, c, s) \wedge Better^*(c, b, s)\big) \tag{A.9}$$

$$Pref(a, b, s) \leftrightarrow Better^*(a, b, s) \vee \big(a <_{alph} b \wedge \neg Better^*(a, b, s) \wedge \neg Better^*(b, a, s)\big) \tag{A.10}$$

The search algorithm associated with the heuristic forward chaining planner described in this paper can be formalized by the strict linear order on situations $HFCP(s_1, s_2, l)$, which is true for a pair of situations s_1, s_2, and a depth limit l, if situation s_1 is explored before situation s_2 by the planner when it is called with depth limit l. The strict linear order $HDFS(s_1, s_2)$ modifies the order associated with depth first search (Axiom (A.4)) by using the preference relation on actions $Pref(a, b, s)$, which takes into account the heuristic information provided by the predicate $Better(a, b, s)$. The strict linear order associated with the planner $HFCP(s_1, s_2, l)$ is simply a restriction of the order relation $HDFS(s_1, s_2)$ to the set of selectable situations according to a given depth limit and the heuristic knowledge provided by the predicates $Good(a, s)$ and $Bad(a, s)$.

$$HDFS(s_1, s_2) \leftrightarrow s_1 < s_2 \vee \exists s, a, b\big(Result(a, s) \leqslant s_1 \wedge Result(b, s) \leqslant s_2 \wedge Pref(a, b, s)\big) \tag{A.11}$$

$$HFCP(s_1, s_2, l) \leftrightarrow Sel(s_1, l) \wedge Sel(s_2, l) \wedge HDFS(s_1, s_2) \tag{A.12}$$

The first solution returned by the planner when it is called with depth limit l can be characterized as follows. $G(s)$ is a predicate that is true if situation s satisfies the conditions of the goal for which a plan is sought, and r is a variable of the sort action sequence.

$$\forall s(Result([], s) = s) \wedge \forall a, s\big(Result([a|r], s) = Result\big(r, Result(a, s)\big)\big) \tag{A.13}$$

$$First(r, l) \leftrightarrow \exists s\big(s = Result(r, S_0) \wedge Sel(s, l) \wedge G(s) \wedge \neg\exists s_1\big(G(s_1) \wedge HFCP(s_1, s)\big)\big) \tag{A.14}$$

The axiom set $T_{HFCP} = \{(A.1), \ldots, (A.3), (A.5), \ldots, (A.14)\}$ constitutes a formal model of the heuristic forward chaining planner described in this paper.

We describe below the sort of information that must be provided to the heuristic forward chaining planner in order to solve a planning problem. This information consists of: (1) *domain knowledge*, a description of the dynamics of a planning domain; (2) *a strategy for action selection*, a consistent set of action selection rules describing which actions are good, bad, or better than others for a particular situation; and (3) *a problem description*, a specification of a problem instance to be solved.

A.1. Domain knowledge

The following action precondition and successor state axioms [35] describe the basic theory of action of the blocks world assumed by the action selection strategy presented in Section 2.1. T is a constant symbol which denotes the table.

$$Poss(M(x, y, z), s) \leftrightarrow x \neq T \wedge x \neq z \wedge On(x, y, s) \wedge Clear(x, s) \wedge Clear(z, s) \tag{A.15}$$

$$Poss(a, s) \rightarrow \big(On\big(x, y, Result(a, s)\big) \leftrightarrow \exists z\big(a = M(x, z, y)\big) \vee \big(On(x, y, s) \wedge \neg\exists z(a = M(x, y, z))\big)\big) \tag{A.16}$$

$$Poss(a, s) \rightarrow \big(Clear(x, s) \leftrightarrow x = T \vee \exists y, z\big(a = M(y, x, z)\big) \vee \big(Clear(x, s) \wedge \neg\exists y, z\big(a = M(y, z, x)\big)\big)\big) \tag{A.17}$$

A.2. Action selection strategy

The action selection strategy for the blocks world described in Section 2.1 uses some concepts which are not usually included in basic theories of action of the blocks world. The formal definitions of such concepts are described below. The axiom set $T_{BW} = \{(A.15), \ldots, (A.23)\}$ is a formal model of the extended theory of action used by the planner in order to reason about the dynamics of the blocks world.

$$Ong(x, y) \leftrightarrow \forall s\big(G(s) \rightarrow On(x, y, s)\big) \tag{A.18}$$

$$Above(x, y, s) \leftrightarrow On(x, y, s) \vee \exists z\big(On(x, z, s) \wedge Above(z, y, s)\big) \tag{A.19}$$

$$Aboveg(x, y) \leftrightarrow \forall s\big(G(s) \rightarrow Above(x, y, s)\big) \tag{A.20}$$

$$Final(x, s) \leftrightarrow \big(Ong(x, T) \wedge On(x, T, s)\big) \vee \exists y\big(y \neq T \wedge Ong(x, y) \wedge On(x, y, s) \wedge Final(y, s)\big) \tag{A.21}$$

$$TowerD(x, s) \leftrightarrow \exists y\big(Aboveg(x, y) \wedge Above(x, y, s) \wedge y \neq T \wedge \neg Final(x, s)\big) \tag{A.22}$$

$$NextFinal(x, s) \leftrightarrow \neg Final(x, s) \wedge \exists y\big(Ong(x, y) \wedge \big(y = T \vee (Clear(y, s) \wedge Final(y, s))\big)\big) \tag{A.23}$$

The axiom set $S_{BW} = \{(A.24), \ldots, (A.28)\}$ is a formal description of the action selection strategy for the blocks world described in Section 2.1.

$$On(x, y, s) \wedge Clear(x, s) \wedge \neg Final(x, s) \wedge Ong(x, z) \wedge \big(z = T \vee \big(Clear(z, s) \wedge Final(z, s)\big)\big) \rightarrow Good\big(M(x, y, z), s\big)$$
$$On(x, y, s) \wedge Clear(x, s) \wedge TowerD(x, s) \rightarrow Good\big(M(x, y, T), s\big) \tag{A.24}$$

$$Final(y, s) \wedge Ong(y_2, y) \wedge \big(\neg\exists w_2 Ong(w_2, w) \vee \neg Final(w, s)\big) \rightarrow Better\big(M(y_1, y, T), M(w_1, w, T), s\big) \tag{A.25}$$

$$NextFinal(y, s) \wedge \neg NextFinal(v, s) \rightarrow Better\big(M(y_1, y, T), M(v_1, v, T), s\big) \tag{A.26}$$

$$Final(x, s) \rightarrow Bad(M(x, y, z), s) \tag{A.27}$$

$$Ong(x, w) \wedge \big(\neg Clear(w, s) \vee \neg Final(w, s)\big) \wedge z \neq T \rightarrow Bad\big(M(x, y, z), s\big) \tag{A.28}$$

Action selection strategies are interpreted nonmonotonically using *predicate completion* [7]. The result of applying the completion algorithm described below to the formal specification of a strategy for action selection is a set of formulas which characterize the extensions of the predicates *Good*, *Bad* and *Better*. Let T_S be the formal description of an action selection strategy such that the antecedent of every action selection rule in T_S does not contain any instance of the predicates *Good*, *Bad* or *Better*. The completion of T_S is the set of formulas (A.29), where $A_3^{good}(a, s)$, $A_3^{bad}(a, s)$ and $A_3^{better}(a, b, s)$ are as described in the completion algorithm below.

$$\begin{aligned} \mathrm{COMP}(T_S) \equiv \big\{&\forall a, s\big(Good(a, s) \leftrightarrow A_3^{good}(a, s)\big), \\ &\forall a, s\big(Bad(a, s) \leftrightarrow A_3^{bad}(a, s)\big), \\ &\forall a, b, s\big(Better(a, b, s) \leftrightarrow A_3^{better}(a, b, s)\big)\big\} \end{aligned} \tag{A.29}$$

A.2.1. Completion algorithm

Let T_S be a declarative formalization of a strategy for action selection. The axioms of T_S are all of the form $A \rightarrow P(\vec{t_a}, t_s)$, where A is a first order formula which does not contain the predicates *Good*, *Bad* or *Better*, $\vec{t_a}$ is a tuple of terms of the sort *action*, t_s is a term of the sort *situation*, and P is one of the predicates *Good*, *Bad* or *Better*.

Step 1 Replace each rule of the form $A \to P(\vec{t_a}, t_s)$ in T_S by $A \wedge \vec{a} = \vec{t_a} \wedge s = t_s \to P(\vec{a}, s)$, where $\vec{a}$ is a tuple of new variables of the sort *action*, and s is a new variable of the sort *situation*.

Step 2 Replace each rule $A_1(\vec{a}, s) \to P(\vec{a}, s)$ obtained in the previous step by $\exists \vec{x} A_1(\vec{a}, s) \to P(\vec{a}, s)$, where $\vec{x}$ are the free variables in the original rule.

Step 3 For each P, replace all the rules of the form $A_2^i(\vec{a}, s) \to P(\vec{a}, s)$ obtained in step 2 by a single rule of the form $\bigvee_i A_2^i(\vec{a}, s) \to P(\vec{a}, s)$.

Step 4 Replace the rules

$$A_3^{good}(a, s) \to Good(a, s), \quad A_3^{bad}(a, s) \to Bad(a, s) \quad \text{and}$$
$$A_3^{better}(a, b, s) \to Better(a, b, s)$$

obtained in step 3 by

$$\forall a, s\big(Good(a, s) \leftrightarrow A_3^{good}(a, s)\big), \quad \forall a, s\big(Bad(a, s) \leftrightarrow A_3^{bad}(a, s)\big) \quad \text{and}$$
$$\forall a, b, s\big(Better(a, b, s) \leftrightarrow A_3^{better}(a, b, s)\big),$$

respectively.

A.3. Problem description

Axioms (A.30)–(A.32) describe the initial and goal configurations of a planning problem P5. Axiom (A.33) is a domain closure axiom. $B_1, \ldots, B_5$ are constant symbols of the sort block. $T_{P5} = \{(A.30), \ldots, (A.33)\}$ is a formal description of blocks world problem P5.

$$Clear(x, S_0) \leftrightarrow x = B_1 \vee x = B_3 \tag{A.30}$$

$$On(x, y, S_0) \leftrightarrow (x = B_1 \wedge y = B_5) \vee (x = B_5 \wedge y = B_2) \vee (x = B_2 \wedge y = T) \vee (x = B_3 \wedge y = B_4) \vee (x = B_4 \wedge y = T) \tag{A.31}$$

$$G(s) \leftrightarrow On(B_5, B_1, s) \wedge On(B_1, B_4, s) \wedge On(B_4, T, s) \wedge On(B_3, B_2, s) \wedge On(B_2, T, s) \tag{A.32}$$

$$\forall x(x = B_1 \vee x = B_2 \vee x = B_3 \vee x = B_4 \vee x = B_5 \vee x = T) \tag{A.33}$$

The formal model of the heuristic forward chaining planner described in this paper T_{HFCP}, together with the action theory for the blocks world T_{BW}, the description of the problem T_{P5}, and the completion of the strategy for action selection S_{BW} imply that the first solution returned by the heuristic forward chaining planner for problem P5 is $\{M(B_3, B_4, T),\ (B_1, B_5, B_4),\ M(B_5, B_2, B_1)\ , M(B_3, T, B_2)\}$.

Appendix B. Action selection strategy for logistics

The strategy for action selection used in the experiments for the domain of *logistics* is reproduced below. It is important to notice that this strategy uses a single action `m(OS,V,O,D)`, which moves a set of objects `OS` from location `O` to location `D` using vehicle `V`, instead of four actions of the form `load, unload, drive` and `fly`, as the

strategies used by other planners do. This formulation of the domain of logistics has the advantage of reducing significantly the size of the search space and allowing the generation of plans with less actions.

The strategy consists of a set of heuristics, represented by action selection rules, which are described below. Each heuristic makes use of some concepts whose definitions are presented together with the description of the heuristic. The basic predicates used in the definition of the problem are: `truck`, `airport`, `package` and `location`, which are true for their arguments if they belong to the corresponding categories; `at(V,O,S)`, which is true for a vehicle `V` if it is at location `O` in situation `S`; and `in_city(L,C)`, which is true if location `L` is in city `C`. Finally, the symbol `\+` is used to represent the negation as failure Prolog operator.

Heuristic 1. If the only objects that must be moved in a city are at post office O or at airport D, there is a truck V at O, and there is an object at O that must be at D or at another city, then it is good to move from O to D with truck V all the objects that must be at different locations.

```
good(m(OS,V,O,D),S) :- truck(V), at(V,O,S),
     \+ airport(O), in_city(O,C), in_city(D,C),
     airport(D), at(X,O,S), package(X), (at(X,D,sg) ;
     other_city(X,C)), \+ move_others(O,D,S),
     no_final_set(OS,O,S).

/* Package X is at O in situation S, but this is not its
final position. */

no_final(X,O,S) :- at(X,O,S), package(X), \+ at(X,O,sg).

/* OS is the set of packages that are at O in
situation S but this is not their final position. */

no_final_set(OS,O,S) :- findall(X,no_final(X,O,S),
                                    OS), !.

/* There are packages that are not in their final
positions at places of the city different from O and
D. */

move_others(O,D,S) :- in_city(O,C), in_city(L,C),
    \+ O=L, \+ D=L, at(X,L,S), package(X),
    \+ at(X,L,sg).

/* X must be in a city different from C. */

other_city(X,C) :- at(X,L,sg), in_city(L,C1), \+ C=C1.
```

Heuristic 2. If a city is complete at a given situation,[6] all the objects in the city except those at airport O are in final position, there is a truck V at O, and some of the objects at O must be at another location D of the city, then it is good to move from O to D with truck V all the objects that are at O in situation S but must be at different locations of the same city.

```
good(m(OS,V,O,D),S) :- truck(V), at(V,O,S), airport(O),
     in_city(O,C), complete_city(C,S), at(X,O,S),
     package(X), at(X,D,sg), in_city(D,C), \+ D=O, !,
     all_final_in_city_but(O,S),
     findall(Y,same_city_diff_location(Y,C,O,S),OS).

/* A city C is complete in situation S if all the
objects that must be at C are at some location of C
in situation S. */

complete_city(C,S) :- \+ not_complete_city(C,S).

not_complete_city(C,S) :- in_city(L,C), at(X,L,sg),
    package(X), at(X,M,S), in_city(M,D), \+ C=D, !.

/* All the objects that are at city C in situation S are
in final position except those that are at location
L. */

all_final_in_city_but(L,S) :- \+ move_others(L,L,S).

/* Package X is at O in situation S, but must be at a
different location of the same city. */

same_city_diff_location(X,C,O,S) :- at(X,O,S),
    package(X), at(X,L,sg), in_city(L,C), \+ O=L.
```

Heuristic 3. If all the objects in the city except those at airport D are in final position, there is an object at D that must be at another location of the city and there is no truck at D, then it is good to move truck V from O to D.

```
good(m([],V,O,D),S) :- airport(D),
    all_final_in_city_but(D,S),
    in_city(D,C), same_city_diff_location(_,C,D,S),
    \+((truck(V1), at(V1,D,S))), in_city(O,C), \+ O=D,
    at(V,O,S), truck(V).
```

[6] A city is complete at a given situation if all the objects that must be in that city are at some location of the city in that situation.

Heuristic 4. If all the objects that are in a city C at situation S and must not be in C, and all the objects required in another city T are at an airport O of C, then it is good to move from O to an airport D of city T all the objects that are at O in situation S but must not be in city C.

```
good(m(OS,V,O,D),S) :- airplane(V,O,S), in_city(O,C),
    \+ ((must_be_in_diff_city(X,C,S), \+ at(X,O,S))),
    findall(X,other_city(X,O,C,S),OS),
    airport(D), \+ D=O, in_city(D,T), \+ C=T,
    findall(X,must_be_in_city(X,T,S),OC), \+ OC=[],
    list_to_ord_set(OS,OL), list_to_ord_set(OC,OD),
    ord_subset(OD,OL).

/* Airplane V is at airport O at situation S. */

airplane(V,O,S) :- airport(O), at(V,O,S), \+ truck(V),
    \+ package(V).

/* Package X must be at city C, but it is at another
city in situation S. */

must_be_in_city(X,C,S) :- in_city(L,C), at(X,L,sg),
    package(X), at(X,L1,S), in_city(L1,D), \+ C=D.

/* Package X is at some location of city C in situation
S, but must be in another city. */

must_be_in_diff_city(X,C,S) :- in_city(L,C), at(X,L,S),
    package(X), other_city(X,C).

/* Package X is at location O of city C in situation S,
but must be at another city. */

other_city(X,O,C,S) :- at(X,O,S), in_city(O,C),
     package(X), other_city(X,C).
```

Heuristic 5. It is bad to move a truck from airport O to post office D if there is nothing to deliver and nothing to pick up.

```
bad(m(_,V,O,D),S) :- airport(O), at(V,O,S), truck(V),
    in_city(O,C), in_city(D,C), \+ O=D, \+ airport(D),
    \+ no_final(_,D,S), \+ deliver_po(O,D,S).

deliver_po(O,D,S) :- at(X,O,S), package(X), at(X,D,sg).
```

Heuristic 6. It is bad to move a truck from post office O to airport D if there is nothing to deliver, and there is a truck at D or nothing to pickup.

```
bad(m(_,V,O,D),S) :- in_city(O,C), \+ airport(O),
    at(V,O,S), truck(V), \+ no_final(_,O,S),
    in_city(C,D), airport(D), at(T,D,S), truck(T).

bad(m(_,V,O,D),S) :- in_city(O,C), \+ airport(O),
    at(V,O,S), truck(V), \+ no_final(_,O,S),
    in_city(D,C), airport(D),
        \+ same_city_diff_location(_,C,D,S).
```

Heuristic 7. It is bad to move a truck from airport O to post office D, if the city is not complete (i.e., there are some objects that must be in the city but are at different cities in situation S) and there is nothing to pickup at D.

```
bad(m(_,V,O,D),S) :- airport(O), at(V,O,S), truck(V),
    in_city(O,C), not_complete_city(C,S), in_city(D,C),
    \+ airport(D), \+ no_final(_,D,S).
```

Heuristic 8. It is bad to move a set of objects OS from an airport O of a city C with a plane V if OS is not the set of objects that are in city C at situation S but must be in another city.

```
bad(m(OS,V,O,_),S) :- airplane(V,O,S), in_city(O,C),
    findall(X,must_be_in_diff_city(X,C,S),GO),
    list_to_ord_set(OS,OL), list_to_ord_set(GO,GL),
    \+ ord_seteq(OL,GL).
```

Heuristic 9. It is bad to move an airplane from airport O to airport D if there is nothing to deliver, and nothing to pick up at D or there is an airplane at D.

```
bad(m(_,V,O,D),S) :- airplane(V,O,S), in_city(D,C),
    \+ deliver_a(O,D,S), \+ pickup_a(D,C,S).

bad(m(_,V,O,D),S) :- airplane(V,O,S),
    \+ deliver_a(O,D,S), airplane(_,D,S).

/* There is something to deliver from airport O to
airport D, if there is an object at O that must be
at the city where D is located. */

deliver_a(O,D,S) :- at(X,O,S), package(X), at(X,L,sg),
    in_city(L,C), in_city(D,C).

/* There is something to pickup with an airplane at
```

```
airport O, if there is an object at O that must be at
another city. */

pickup_a(O,C,S) :- at(X,O,S), package(X),
other_city(X,C).
```

References

[1] AIPS, Artificial Intelligence Planning Systems: 1998 Planning Competition, 1998, http://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html.

[2] AIPS, Artificial Intelligence Planning Systems: 2000 Planning Competition, 2000, http://www.cs.toronto.edu/aips2000.

[3] F. Bacchus, F. Kabanza, Planning for temporally extended goals, in: Proceedings of AAAI-96, Portland, OR, 1996, pp. 1215–1222.

[4] F. Bacchus, F. Kabanza, Using temporal logic to control search in a forward chaining planner, in: M. Ghallab, A. Milani (Eds.), New Directions in AI Planning, IOS Press, Amsterdam, 1996, pp. 141–153.

[5] F. Bacchus, F. Kabanza, Using temporal logics to express search control knowledge for planning, Artificial Intelligence 116 (2000) 123–191.

[6] B. Bonet, H. Geffner, Planning as heuristic search, Artificial Intelligence 129 (1–2) (2001) 5–33.

[7] K. Clark, Negation as failure, in: H. Gallaire, J. Minker (Eds.), Logic and Data Bases, Plenum Press, New York, 1978, pp. 293–322.

[8] Y. Dimopoulos, B. Nebel, J. Koehler, Encoding planning problems in nonmonotonic logic programs, in: Proceedings of the Fourth European Conference on Planning, 1978, pp. 169–181.

[9] P. Doherty, J. Kvarnstrom, TALplanner: A temporal logic-based planner, AI Magazine 22 (3) (2001) 95–102.

[10] E. Erdem, Applications of logic programming to planning: computational experiments, Technical Report, Computer Science, University of Texas at Austin, 2000, http://www.cs.utexas.edu/users/esra/papers.html.

[11] R. Fikes, N. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, Artificial Intelligence 2 (3–4) (1971) 189–208.

[12] A. Finzi, F. Pirri, R. Reiter, Open world planning in the situation calculus, in: Proceedings of AAAI-00, Austin, TX, 2000, pp. 754–760.

[13] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: Proceedings of the Fifth International Conference on Logic Programming, 1988, pp. 1070–1080.

[14] M. Gelfond, V. Lifschitz, Logic programs with classical negation, in: Proceedings of the Seventh International Conference on Logic Programming, 1990, pp. 579–597.

[15] M. Genesereth, J. Hsu, Partial programs, TR 89-20, Computer Science Department, Stanford University, 1989.

[16] H. Kautz, B. Selman, Blackbox: A new approach to the application of theorem proving to problem solving, Technical Report, 1998. System available at http://www.research.att.com/kautz.

[17] J. Koehler, B. Nebel, J. Hoffmann, Y. Dimopoulos, Extending planning graphs to an ADL subset, in: Proceedings of the European Conference on Planning, 1997, pp. 273–285.

[18] J. Kvarnstrom, P. Doherty, TAL planner: A temporal logic based forward chaining planner, Ann. Math. Artificial Intelligence 30 (1) (2000) 119–169.

[19] H. Levesque, R. Reiter, Y. Lesperance, F. Lin, R. Scherl, GOLOG: A logic programming language for dynamic domains, J. Logic Programming 31 (1997) 59–84.

[20] V. Lifschitz, Answer set planning, in: Proceedings of the International Conference on Logic Programming, 1999, pp. 23–37.

[21] V. Lifschitz, Answer set programming and plan generation, Artificial Intelligence 138 (2002) 39–54.

[22] F. Lin, Applications of the situation calculus to formalizing control and strategic information: The prolog cut operator, in: Proceedings of IJCAI-97, Nagoya, Japan, 1997, pp. 1412–1418.

[23] F. Lin, On measuring plan quality (a preliminary report), in: Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning, Trento, Italy, 1998, pp. 224–232.

[24] F. Lin, Search algorithms in the situation calculus, in: H. Levesque, P. Pirri (Eds.), Logical Foundations of Cognitive Agents: Contributions in Honor of Ray Reiter, Springer, Berlin, 1999, pp. 213–233.
[25] F. Lin, A planner called R, AI Magazine 22 (3) (2001) 73–76.
[26] V. Marek, M. Truszcynski, Stable models and an alternative logic programming paradigm, in: K. Apt, V. Marek, M. Truszczynski, D. Warren (Eds.), The Logic Programming Paradigm: A 25-Year Perspective, Springer, Berlin, 1999, pp. 375–398.
[27] J. McCarthy, Programs with common sense, in: Mechanization of Thought Processes, Proceedings of the Symposium of the National Physics Laboratory, 1959, pp. 77–84, reproduced in [28].
[28] J. McCarthy, Formalizing Common Sense, Ablex, Norwood, NJ, 1990.
[29] J. McCarthy, Lecture notes of CS323: Course on the formalization of common sense and nonmonotonic reasoning, Technical Report, 1997, available at http://www-formal.stanford.edu/jmc/cs323/96/lecture-notes.ps.
[30] J. McCarthy, Notes on human level intelligence, section on heuristics, Technical Report, 1997, available at http://www-formal.stanford.edu/jmc/human/node10.html.
[31] J. McCarthy, Concepts of logical AI, in: J. Minker (Ed.), Logic-Based Artificial Intelligence, Kluwer Academic, Dordrecht, 2001, pp. 37–56.
[32] J. McCarthy, P. Hayes, Some philosophical problems from the standpoint of artificial intelligence, in: B. Meltzer, D. Michie (Eds.), Machine Intelligence, vol. 4, Edinburgh University Press, Edinburgh, 1969, pp. 463–502.
[33] I. Niemela, Logic programs with stable model semantics as a constraint programming paradigm, Ann. Math. Artificial Intelligence 23 (3–4) (1999) 241–273.
[34] E. Pednault, ADL: Exploring the middle ground between STRIPS and the situation calculus, in: Proceedings of the International Conference on Principles of Knowledge Representation (KR-98), Trento, Italy, 1989, pp. 324–332.
[35] R. Reiter, The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression, in: V. Lifschitz (Ed.), Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, Academic Press, New York, 1991, pp. 359–380.
[36] R. Reiter, Proving properties of states in the situation calculus, Artificial Intelligence 64 (2) (1993) 337–351.
[37] R. Reiter, Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, MIT Press, Cambridge, MA, 2001.
[38] J. Sierra, Declarative formalization of strategies for action selection, in: Proceedings of the Seventh International Workshop on Non-monotonic Reasoning, 1998, pp. 21–29.
[39] J. Sierra-Santibáñez, Declarative formalization of reasoning strategies: A case study on heuristic nonlinear planning, Ann. Math. Artificial Intelligence 39 (2003) 61–100.
[40] J. Sierra, Declarative formalization of strategies for action selection: Applications to planning, in: Proceedings of the Seventh European Workshop on Logics in Artificial Intelligence, in: Lecture Notes in Computer Science, Springer, Berlin, 2000, pp. 21–35.
[41] P. Simons, Extending and implementing the stable model semantics, Doctoral Dissertation, Research Report 58, University of Technology, Helsinki, Finland, 2000.
[42] SMODELS 2.26, 2001, available at http://www.tcs.hut.fi/Software/smodels/.
[43] V. Subrahmanian, C. Zaniolo, Relating stable models and AI planning domains, in: Proceedings of the International Conference on Logic Programming, 1995, pp. 233–247.
[44] T. Syrjanen, Implementation of local grounding for logic programs with stable model semantics, B 18, Digital Systems Laboratory, University of Technology, Helsinki, Finland, 1998.
[45] Systems, 2001, Systems capable of computing answer sets are available at: http://www.tcs.hut.fi/Software/smodels, http://www.dbai.tuwien.ac.at/proj/dlv, http://www.cs.engr.uky.edu/~lpnmr/DeReS.html, http://www.cs.utexas.edu/users/mmcain/cc.